Devonfw Methodology

2017-04-18

Table of Contents

1. System Specification

1.1 System Specification Overview

There are many methods to write specifications, especially business-oriented system specifications. As a part of the devonfw methodology module, one such method is described, which is pragmatic, easy to use and open for every team member.

This method is described in the following ways:

Guide
: As a guide to the creation of a system specification and the integration into an agile project methodology

Example
: As an example of a system specification, written for the restaurant example of devonfw.

1.2 Guide to the system specification method

1.2.1 Introduction

In a classical project setup using a *waterfall* methodology, there are early phases, in which the following activities happen:

- business modeling
- requirements engineering
- system specification

They build upon each other. The output of the *system specification* activity is a very detailed specification document that is one of the main inputs for the following phases, in this case the *technical system design* and the *implementation*.

The number of current projects using a waterfall methodology is quite low. Never the less, a system specification provides great value. This guide discusses

- the necessity for such a specification
- the integration into the development methodology
- the form and contents of a system specification

It is based upon experiences in current software projects, combined with knowledge of the traditional specification methodology.

Goal of this guide

This guide is no introduction to UML or to the basic concepts of use cases and system specifications. Please refer to the *Rational Unified Process* for more a more full introduction (see for example: The Rational Unified Process: An Introduction). The purpose of this guide is to adapt this basic methodology so that it is

- as pragmatic and lightweight as it can be.
- near to the actual code and system. Some abstractions will however not be avoidable.
- as easy to do for each participant (especially for each member of the development team) as it can be.
- as short and non-redundant as it can be.

This document is intended to be a very short guide: It provides information in a short and abbreviated form, oriented at developers, architects and business analysts with some experience of their own.

1.2.2 Starting point

The need to specify a feature before implementing it

This guide is based on the assumption that it is important to specify before you code. The level of detail depends on the task and the project methodology, the form and content may vary, but in a professional environment it is necessary to define the business and technical content before it is implemented. The advantages and reasons for this are described throughout this guide.

There are opinions about agile project saying that neither planning nor documentation is needed because of the changing nature of the project: These opinions are wrong. The necessity and effort necessary to plan, to document, to test and generally to assure the quality of the resulting system in an agile project are at least as high in an agile project as they are in a classical waterfall project. This, too, will be discussed throughout this guide.

This does, however, not mean that each project should do a specification in the same way. This guide will provide different examples and best practices to use in a project. It does not provide rules, merely suggestions. It is up to the reader to tailor the approach to his or her needs.

Advantages of a concise and up-to-date functional documentation

Current agile methodologies like SCRUM do not specify the type of documentation to create for a system that is being created: SCRUM defines the contents of a backlog item, but a backlog item is not a part of a specification of the final system: It is the specification of a task that shall be done during a sprint. It may contain technical explorations, refactoring, partial implementations and other topics.

The specification of the final system is typically part of the *definition of done* of all functional backlog items, or the content of a separate backlog item, for the team members: They need to write down the specification, potentially in a wiki, as a basis for future work.

The advantages of this documentation are, among others, the following:

Stable functional architecture
: Teams change, and teams need to maintain and further develop many systems. The functional architecture laid down and documented in the system specification helps to maintain the initial functional architecture of the system.

Ubiquitous language
: This is a concept of the *Domain Driven Design* described by *Eric Evans* (see Domain driven design quickly). It creates a language that is used from the discussions with the business departments (almost) down to the last line of the code. A good system specification is written and structured so that the business parts of it can be coded as they are documented: The business components, the use cases and use case functions should be seen in the code exactly as they are in the specification. Some deviations may exist because of technical reasons: They should be documented in the appendix of the specification itself. This is extremely helpful, especially in discussions with the business departments and in the onboarding of new team members.

Reduced efforts in technical design
: Since e.g. the business components, model components, use cases, entities directly relate to packages and classes in the specification, this information does not need to be defined as part of the technical design.

Reference for discussions
: According to the SCRUM methodology, the results of a sprint are accepted in the sprint review. The discussion done in the sprint review (or later) is concerned with new backlog items to correct any discrepancies between actual and desired behavior. In some projects, this is not the case: The software may in some cases be tested and accepted formally over a longer period after each sprint has finished. Or the sprints may be offered as a fixed price, and later discussions concerning deviations from expected behavior may occur in the acceptance of the deliverable. An accepted specification would be the reference by which these questions concerning acceptance testing and required behavior can judged: User stories will be too coarse grained to help in many of these occasions.

In order to leverage these advantages, the system specification needs to be accepted by both the development team and the customer. It is therefore necessary to **integrate the specification in the project methodology**. This will be discussed later in this document.

Overview of the system specification method

Advantages of using one defined method for system specifications

There are different ways to create a system specification. The advantages to use the same (good and pragmatic) specification method each time are among others:

- Ramp-up times are reduced since the specification method is up and running in less time.
- The system specification can be completed quicker with fewer frictional losses.
- The quality of the system specification increases.

Using the development-oriented specification method defined in this document provides following additional benefits:

- The specifications allows the depending disciplines design, implementation, test, etc. to be processed more quickly and efficiently and with higher quality.
- The method provides a structuring of the system.

Overview of the contents of a system specification

A system specification should provide the following information about a system:

- A functional overview
- A description of the behavior of the system
- A description of the structure of the system
- A definition of the interaction of the system with the user and other systems
- A definition of cross-cutting concepts, e.g. concerning authorization, archiving, logging, multi-language ability.
- Any other required additional information

In order to provide this information, the following artifacts are used in the methodology described in this document.

- **Business components** to structure the behavior of a system
- **Use cases**, **use case functions** and **batches** to define the behavior of the system for the user and for other systems
- **Model components** to structure the data of a system
- **Entities** and **datatypes** to define the data of a system
- **Dialogues** to define the workflows of the user interface
- **Screens** to define the different screens of the user interface and their contents

- **Provided interfaces**, **Used interfaces** and **interface entities** to define the system interfaces.
- **non-functional requirements** to define the non-functional requirements of the system.
- Textual representation for cross-cutting concepts and other required information.

```
TODO: Diagram
```

Method to write a system specification

A system specification is produced in two steps:

- First, a **coarse-grained overview** of the system and its functionality is developed. The most important product of this phase is (a first version of) the functional overview. In this phase, the system is organized into business components and model components, and an overview about the core use cases is created. To do this, workshops are done with the product owner and the business departments. If any requirements for the system exist, they are the basis for the creation of the overview. In a SCRUM project, this is done during the project initialization, before the sprints begin. The result is a skeleton of a specification document filled with empty chapters for the most important artifacts, already structured according to the business and model components.
- Secondly, a **detailed specification** is made, producing the individual artifacts of the system specification, building the complete specification of the system. The specification process is organized according to the previously identified conceptual components. In a SCRUM-project, the way to do this may differ: Different variants are described later in this document.

```
TODO: Flesh out this topic: include aspects of requirements analysis and the process of creating of
 the functional architecture.
```

User Stories and system specifications

User stories are part of the agile SCRUM methodology discussed later. In this overview, only their relationship with system specifications is discussed.

Often, a user story will cover a use case or a part of a use case. But this is not a fixed rule: A user story may cover (parts of) many use cases, and a use case may cover many user stories. There is no direct relation between use cases and user stories.

In SCRUM, a user story should contain enough information to implement it without interruptions. Typically, it will however contain less details than a system specification, since the team will decide on certain aspects themselves during the implementation (e.g. the package and class structure), and since the product owner is part of the team and available for **quick decisions** about details arising during the implementation. If this is the case, user stories are a perfect fit.

User stories are also a great basis for sprint reviews, since this is done in a workshop and as a discussion. It is, however, not best suited if formal acceptance testing is done by a customer (this is, of course, a deviation from the SCRUM methodology, but quite often the case):

- The tests specified by a customer need a precise and detailed basis, which is typically not fully provided by a user story.
- The tests will probably be done multiple times, since the same functionality (the same use case) is enhanced and completed in multiple user stories. If a test is referencing only a story, it needs to be searched, found and associated with the next story that deals with the same use case.

Therefore, if acceptance tests are done by the customer, another kind of documentation should be present at the end of a sprint.

Additionally, user stories are not suited to be a documentation of a system after a project (or a project phase) has finished: They are task-oriented, and do not provided a uniform view of the target system. Therefore, even if no acceptance testing is done, another kind of documentation should be present at the end of project or project phase.

A system specification as described in this document is a good format for such a documentation.

1.2.3 Guiding principles of a system specification

A system specification easily turns out to be huge, cumbersome, hard to extend and only updated by a small number of analysts. Without huge organizational efforts and costs, this will cause the specification to quickly become out of date: There is so much to do and so much stress in each sprint that the analysts will become a bottleneck and will have to prioritize their work.

In order to make it possible to have an up to date system specification:

- each team member must be able to update it
- it must be quickly changeable
- there must be no technical or organizational barriers to the editing of the specification
- the update of the specification must be deeply integrated into the project methodology

Therefore, the following principles are proposed for system specifications in the devonfw context:

Storage together with the code, versioned together with the code
: This reduces the barrier to update the specification, and ensures that the right version of the specification is associated with the right deliverable. If the specification is stored and edited in a wiki, this principle is void: In this case, the ease of use of a wiki outweighs the advantages of the uniform versioning.

Usage of existing tools
: The update of the specification should not require expensive or hard to use products. This concerns mainly the UML tool: It should be a low-level tool which is easy to use and easy to understand. In the example specification of devonfw, plantuml (see PlantUML) is used.

Small size of the specification
: Since the specification may be part of a git repository, it should use little disk space. In the example specification of devonfw, we use the markup language asciidoc. Using a markup language adds the benefit of being easy to merge, if a specification has been edited by more than one person.

  Additionally, the real size of the specification should be small, as is described in the next points.

No irrelevant data
: Everything that is self-explanatory can be omitted. If an entity called "PassportData" contains the attributes "PassportNumber", "IssuingState" and "DateOfExpiry", neither the entity nor the attributes need any documentation concerning their semantics. Relevant are of course the data types, the multiplicities, and the associations. These must be defined, the other documentation may be omitted. The diagram of a data model may therefore already contain nearly all the information needed in the specification.

No redundant information
: The same information should not be described twice in the specification. An example are use cases and dialogues: Typically, the description of functionality is separated between the concrete screens in the dialogue and the behavior as part of a use-case. This e.g. allows the use case to also be automatically accessed via a system interface. If, however, a functionality is clearly only used for a screen or a dialogue, it should be described only once: A use case should be created as an anchor for the implementation of the functionality, but it should reference the dialog for the procedural information.

Usage of external sources
: If there is already an xml schema defining a system interface, it is not necessary to re-document the xml schema in UML as part of a system specification: A documentation of the xsd (e.g. as a generated HTML file) can be used as a supplementary document, and be referenced in the documentation of the system interface.

1.2.4 Integration of the system specification into the development process

This chapter gives an overview about how the creation of a system specification is integrated into the development process.

In the following, two development approaches are discussed as examples for project approaches:

- The **waterfall oriented development process**, see e.g. Waterfall model. Two variants of this method will be discusses: The traditional the incremental waterfall approach.
- The **agile development process**. The integration will be discussed for different variants of the SCRUM methodology (see e.g. SCRUM)

The methodology is, however, not the core part concerning the system specification: A system specification provides added value in nearly all project approaches. The core aspect remains the necessity to **speficy a feature before coding it**, independent of the concrete approach.

If you are not interested in the integration of the specification in the project methodology, please skip to the chapter .

Integration of a specification into a waterfall oriented development process

```
TODO: Discuss the usage in a waterfall or an incremental waterfall development process. This is pretty
 straight forward and should be very near to the variant "specification first" description in the agile
 development process.
```

Integration of a specification into an agile development process

Variant: specification first

```
TODO: Diagram
```

The most straight forward way to write a specification is to write it upfront before the sprint cycles. The main advantages of this approach are:

- The possibility to use the specification as a basis for a tender
- A large amount of time can be used for the review and acceptance of the specification: During the sprint cycles, the review of stories has to be done fairly quickly, and may be overlapping with the review of the past sprint: There is little time to really think about the specified system and to discover alternatives.

- Reduction of the workload of the product owner and the business departments during the sprint cycles: This workload is typically high, since the specification has to be done in parallel to the review activities and the solution of business problems.
- Review in one go: Instead of reviewing functional specifications in different states of completeness, the specification is reviewed just once, in a final stage.
- Good planning: Based on this specification, a very detailed backlog with excellent effort estimations is possible. The project is plannable in detail with a much farther horizon than the traditional agile project.

The writing of a full specification up front has several preconditions and disadvantages:

- The full functionality of the system must be known in detail at the beginning of the project: This is a characteristic of a waterfall project an typically not the case in agile projects. However, there are some projects for which this approach is fitting: Examples are reengineering projects, in which one software is replaced by another one with similar functionality. Or the implementation of a law that clearly defines the requirements of the systems.
- To specify up front typically makes the project a bit longer, since the development team needs to start later. This difference is however (according to experience) not that big.
- The acceptance of a specification is harder for the business departments than the acceptance of stories, since it is much larger, and the fragmentation of the content in the document makes it harder to review.

If the specification is done first, the implementation and test phases should still be done in the form of sprints:

- Stories should be created on the basis of the specification, typically referencing the parts of the artifacts they are implementing
- Sprints should be planned, executed, and reviewed. Retrospectives should be done
- Necessary changes to the specification should be identified during the sprints, and the specification shall be changed if necessary.

This kind of project approach provides the advantages of early system releases and early benefit for the customer, together with the ability to accurately plan the milestones and contents of the project beforehand. There are, however, few projects that allow such an approach.

Variant: Agile team with boundaries between development team and product owner

```
TODO: Diagram
```

In many projects, the product owner and the development team are not part of the same company or department: The development team may be part of a custom software development company while the product owner is an employee of the customer. In these cases, the sprint review may be supplemented by a more formalized acceptance test of the customer. The individual tests for the acceptance are specified beforehand by the customer, or by another contractor responsible for quality assurance.

If there are acceptance tests, and especially if the acceptance tests are written by people who are not part of the sprint team, a detailed specification is necessary in order to provide the quality assurance team with a basis to create their test cases on, and to provide a basis to decide on any dispute about delivered vs. expected behavior.

In these cases, the definition of a story (a story document) should contain this detailed specification, and the specification should use the artifacts of a system specification: Business Components, Use Cases, Entities, System interfaces etc. This allows the tests to be based upon these artifacts, and to reuse the tests when the artifacts are extended or changed.

Since the use of such artifacts is harder to understand for the involved people, the story document should also contain an introduction, in which the user story is narrated in brief.

A story document may therefore contain the following elements:

- A short version of the **user story**, describing the goal of the story and the rough contents
- The **artifacts of a system specification**, probably even in the chapter structure of a system specification. If an artifact is extended or changed, the old contents are marked in a different color from the new and the changed contents.
- It proved helpful to also add **hints for the development team**, e.g. concerning changes to the artifacts in future sprints, or necessary structures in the code. This is, however, only an issue in big or distributed teams: Otherwise, the information is provided in enough detail during the sprint planning.

An advantage of such a story document is, that a system specification can easily be created by copying artifacts defined in the story document into the system specification document. And since the contents of the story document have already been accepted as part of the sprints, no elaborate acceptance of the system specification is necessary.

While this approach provides quite a lot of benefits for the development, the tests and the documentation, it is not fit for all projects: It requires a more abstract thinking and more effort on the side of the product owner and the testers. Therefore, the approach needs to be carefully discussed and tested, in order to prevent friction with the product owner or the business department.

Variant: Agile team as described in SCRUM methodology

```
TODO: Diagram
```

In a traditional SCUM project, a user story is written in a business value oriented and narrative way: It will probably be defined in a ticket system, and contain less detail than a system specification, since the product owner is part of the team and can answer any remaining questions quickly.

However, at the end of a project or project phase, some documentation about the system is needed:

- A new project may need to enhance the system, and may need an up to date business specification to do so
- New team members may need to be onboarded, and need an overview over the system
- Errors or deviations from desired behavior may need to be resolved.

To support these activities, some kind of specification is needed. It will probably be part of the DoD of some or even of all stories. Since wikis and ticket systems are used intensively in such a project, the specification will probably be written in a wiki.

Since the specification is written after the system has been coded, it does not need to provide the level of detail necessary in the first two variants: A lot of details (e.g. attributes in entities) can be omitted, in order to reduce the maintenance effort of the specification. The level of detail needs to be discussed and defined per system.

The structure of the system specification method discussed in this document should, however, be used for all of these specification, since it provides a clear and exact documentation mirrored in the code.

The *coarse graining* of a specification mentioned above may also happen in the first two variants after a project or project phase has finished, in order to reduce the maintenance effort of the documentation.

1.2.5 Artifacts of a system specification

```
TODO: Continue from here
```

```
TODO: Focus on concrete examples, bring lots of do's and dont's, do not focus on differences in
 methodologies.
```

Overview

Best Practice: The use of prefixes for terms

If a system is described in detail, the description should be precise: The terms used in the description should be consistent over all chapters of the specification. They should be easily recognizable, both for their type (e.g. use case, entity or attribute) and for the concrete artifact they identify.

Because of this, it is very helpful to use prefixes for the terms used in a system specification.

Examples for this are:

Description	Prefix	Example
An actor (either a person or a technical system) interacting with the specified system	ACT	ACT_Waiter
A use case component separating the behavior of the specified system	UCC	UCC_Statistics
A use case defining the behavior of the specified system	USC	USC_Assign_table
An entity of the data model	ETY	ETY_Waiter
An attribute of an entity	ATT	ATT_Last_name

Best Practice: The use of one word for terms

…

Best Practice: Starting with the interfaces

…

Best Practice: Overview and principles at the beginning

…

Best Practice: Component owns its data

…

Best Practice: No redundancies

```
e.g. Duplicate use cases for manual use and automatic interfaces
```

Best Practice: Coarse grained artifacts

…

[[system-specification-guide_best-practice:-rapid-design-&-visualization]] ==== Best Practice: Rapid Design & Visualization

…

Best Practice: Short descriptions

…

Best Practice: The value of diagrams

…

Best Practice: One document per system

…

Best Practice: Abstract from technical details if necessary

…

Functionality: Use Cases, Use Case functions, Batches

Use cases

Exemplary use case. A use case could for example be written in the following form:

Use Case USC_Assign_table

Description	This use case allows a staff member to assign a table to a waiter.
Actors	ACT_Waiter, ACT_Chief
Usage	Manual, many times per day
Preconditions	• The user has selected a table (an instance of ETY_Table)
Postconditions	• The selected ETY_Table is assigned to a waiter (ETY_Waiter). • The table is not assigned to a waiter: The field waiter is empty

Standard workflow

1. The user chooses to assign a table (TODO: Ref. to datamodel)

2. The system reads and presents a list of the currently active waiters (StaffMember with role "waiter") and proposes to the user to cancel the assignment, to delete the current assignment or to assign a waiter. The deletion of an assignment is only proposed if a waiter is currently assigned to the table.
3. If the user cancelles the assignment: The system returns to the previous screen without changes. The use case is finished.
4. If the user deletes the current assignment of a waiter:
   a. The system removes the assigned waiter form the current table.
   b. The system shows the updated contents of the previous screen.
5. If the user selects a waiter and assigns him to the table:
   a. The system stores the ID of the waiter for the current table.
   b. The system returns to the previous screen
6. The use case is finished

Best practice: active sentences

…

Best practice: Only goals of use case in description text

…

Best practice: Be precise

…

Best practice: Few alternative scenarios

…

Best practice: Coarse grained use cases

…

Data: Entities and datatypes

User interface: Dialogues, screens and print output

System interface: provided interfaces, used interfaces

Non-functional requirements, cross-cutting concepts

The evolution of artifacts over successive sprints

1.2.6 Tooling

```
TODO: Discuss different tools including advantages and disadvantages.
```

1.3 1 Draft for the structure of the MyThaiStar specification

1.3.1 1.1 Basic ideas

The goal of the specification guide and the restaurant specification is to create a document that is:

??? Aligned to the implementation: Structure, concepts and nomenclature should be the same in the code. Since this is an angular client, the whole process flow is part of the client: The server will present REST interfaces for necessary server interaction. These small operations will be documented as use cases and use case functions. The client modules and routes will be documented as part of the dialogue. ??? Allow for compact design documentation: Much of the structuring of components and data should be presented here and not need to be formally repeated in another documentation. The split of the application into components presented here should be present in the code. ??? Slim: It should contain only the strictly needed information and should be very pragmatic. ??? Easy to maintain: It should be versioned together with the code. A developer should be able to change it using only a text editor. Because of these goals, it is very important for me to have a hard discussion about the structure of the components, the data, the use cases and the dialogue features. I am no angular expert and would welcome this.

1.3.2 1.2 Structure of the document

Roughly, the document shall contain the following chapters: ??? Introduction and fundamental ideas ??? Application components, use cases and use case functions ??? Data model ??? Dialogue, Dialogue modules and screens ??? System interfaces ??? Appendix

I do not think we need to specify printouts or non-functional requirements. Otherwise, they would be handled in additional chapters. The following list of contents is a current very first draft and will probably change in the future.

1.3.3 1.3 Structure for the application components and use cases

1.3.1 ACO_Booking_Ordering

This application component handles the management of bookings (whole reservations) and orderings (food and drink for one person).

UC_Book_Table: This use case will get the relevant data for the booking of a table (date, time, email, numbers, friends, invitation text) and create a booking (if possible) for it. If no tables at all are available, it will create an appropriate message. It will send a confirmation email to the host. The invitation of friends could be delegated to UF_Invite_friends.

UC_Order_Meal: This use case will get the relevant data for a meal (dishes, drinks, additions, comments), will validate the data and will store it for one guest of a booking. It will compute the price of the meal including VAT. The system will send an email containing details about the order to the guest.

UF_Invite_friends: This use case function will create unique references (text and QR-Code) for each invited friend and send emails containing an invitation text, a link to directly place the order for the meal, a link to accept the invitation without ordering and a link to refuse. The handling of the link to directly order the meal will be part of a dialogue route, the others will be handled by the following UC. Additionally, a mail containing all relevant information will be sent to the host.

UC_Handle_invitation_feedback: This use case will handle refusals and acceptances for invitation. The UC will validate the data and update the booking. In case of invalid data, it will present an appropriate

response. If a guest has cancelled an invite, all other guests are sent an information, and all previously ordered meals for the guest are deleted.

UC_Select_Table: This use case will be automatically executed by the system after all invitees of a booking have responded or a configured interval before the meal is reached: The system will optimize the table allocation in the restaurant.

UC_Check-In_Guest: This use case will be used when the host or one of the invitees arrive at the restaurant. It will provide information about the reserved table, and it will update the booking status.

UC_Administrate_Booking: This use case allows the creation, update and deletion of bookings and orderings. It will probably be generated via Cobigen.

1.3.2 ACO_Menu_Management

This application component is responsible for managing the data about dishes and drinks the restaurant can offer. The presentation of the menu and the selection of the dishes is done in the dialogue.

UF_Search_Menu-Items: This use case allows to search or filter menu items based on different search parameters, e.g. names, categories, likes or hashtags. A hit list will be returned, the entries of which can then (including images) be read in full.

UF_Administrate_Menu-Items: This use case allows the creation, update and deletion of dishes, drinks, additions, Categories of dishes and drinks, and twitter data (hashtags). It will probably be generated via Cobigen.

1.3.3 ACO_Twitter_Integration

This application component will typically contain little data. It is responsible to encapsulate the handling of the twitter API.

UF_Get_Twitter_Feedback: This use case function uses the configured hashtag of a dish or drink to get the twitter feedback information (number of likes, last comments).

UC_Rate_Dish_Drink: This use case allows to either ???like??? a dish or drink, or to enter a comment for the dish or drink. Both actions will probably create a tweet for the corresponding hashtag.

UC_Update_Rating_Dish_Drink: Since the number of likes is a filter criterion for a dish or drink, the number needs to be updated regularly. This can either be done during UC_Rate_Dish_Drink, or in regular intervals using the twitter API (which would be more stable). In this case, the current use case would check the likes of each dish and drink and update the menu data accordingly.

1.3.4 ACO_User-Data_Management

This application component will manage information about the people using the MyThaiStar application. Currently, this is limited to the twitter data. It could however easily be extended.

UC_Authenticate_User: This use case will authenticate a user based on a user-name and a password. After this, the user will be associated with the current session, and the twitter data will be used for twitter integration.

UF_Get_User-Data: This use case function gets the user data of an already authenticatd user. Currently, it will return the twitter data of the user (see MCO_User-Data).

UC_Manage_User: This use case will handle the update of passwords, of other user data like first and last names, of the twitter data, and the deletion of a user account.

1.3.4 1.4 Structure of the data model

Components encapsulate the data they are responsible for. Therefore, the cutting of components depends upon the data that needs to be managed, and the cutting of data models depends on the components. The data model is therefore cut into two model components: MCO_Bookings_Orders and MCO_Menu.

1.4.1 MCO_Bookings_Orders

Figure 1: Data model structure for booking (not final)

It is important to recognize that every person (friends and host) is a booked person. Even if no email addresses (and only numbers of guests) were given, references for each person will be generated, and ???order-links??? for all participants could be sent to the host.

The order encapsulates the dishes and drinks one participant has ordered:

Figure 2: Data model structure of an order (not final)

The main distinction here is that an ordered dish may contain a comment and a number of additions, while a drink is simply ordered without any supplementary information.

1.4.2 MCO_Menu

The structure of the menu data could be as follows:

Figure 3: The structure of the menu data (not final)

Main decisions here were e.g. that a drink is not specific for a course, and that the categories for drinks and dishes are separated.

1.4.3 MCO_User-Data

This model component encapsulates the user data stored in the MyThaiStar application. Currently, it is limited to the twitter data of the user. It could however also contain favorite dishes or e-mail addresses of persons often invited.

Figure 4: The structure of the user data (not final)

1.3.5 1.5 Structure of the dialogue

I am no expert on angular. I know the dialogue is split into modules, and the dialogue flow is managed as routes. I am not sure on the best way to structure the dialogue specification to allow for a good alignment to the implementation. One way could be a split into the following ???modules??? (instead of screens):

DIM_MyThaiStar_Main: This module includes the welcome page and any additional page (imprint etc.) needed.

DIM_Table_Booking: This module includes the Booking of a table and the entering of e-mail addresses for friends, but not yet the ordering of dishes and drinks.

DIM_Dishes_Drinks_Ordering: This module is perhaps the biggest and contains the presentation of the menu, the search and filter functionality, the twitter integration, the selection and commenting of dishes and drinks, the selection of additions, and the completion of the order.

DIM_Twitter_Rating: This small modules encapsulates the leaving of twitter comments for dishes and drinks. It is perhaps too small and too closely related to the Dishes/Drinks-Ordering Module, and should be integrated into this one. DIM_Account_Management: This module contains the authentication of a user, and the management of a user account: The user can both change the account data and delete the account.

DIM_Menu_Administration: This module should be used for the administration of dishes, drinks, additions, categories, keywords, courses and twitter data. As far as I know, this should be done in a generative approach using Cobigen. DIM_Booking_Administration: This module should be used for the administration of bookings and orders. As far as I know, this should be done in a generative approach using Cobigen.

DIM_Waiter_Cockpit: This Module contains screens only a waiter can use: It shows him a list of the current reservations. For each reservation, all bookings are shown. Optionally, a list of all future bookings could be presented. All data of the reservations and bookings should be seen in this module. The data should not be changeable.

1.3.6 1.6 Structure of the system interfaces

Since the only really external interface is the twitter API and the mail API, no specification seems to be needed here. We could present a vision of the internal REST APIs, but this is best left to the developer, and it is not needed for the client split in one application of the current size: I would omit it here.

2. Client Architecture

2.1 OASP / Devon Client Architecture

2.1.1 Introduction

Purpose of this document

In our business applications, the clients often are much more complex to develop and design than the server. Nonetheless, where we have a concrete layered technical architecture for the server in OASP, we are still lacking a pendant on the client, where we only define to have something called dialog components. Finding an concrete architecture applicable for all clients may on the other hand be difficult to accomplish.

This document tries to define on a high abstract level, a reference architecture which is supposed to be a mental image and frame for orientation regarding the evaluation and appliance of different client frameworks. As such it defines terms and concepts required to be provided for in any framework and thus gives a common ground of understanding for those acquainted with the reference architecture. This allows better comparison between the various frameworks out there, each having their own terms for essentially the same concepts. It also means that for each framework we need to explicitly map how it implements the concepts defined in this document.

The architecture proposed herein is neither new nor was it developed from scratch. Instead it is the gathered and consolidated knowledge and best practices of various projects (s. References).

Goal of the Client Architecture

The goal of the client architecture is to support the non-functional requirements for the client, i.e. mostly maintainability, scalability, efficiency and portability. As such it provides a component-oriented architecture following the same principles listed already in the OASP architecture overview. Furthermore it ensures a homogeneity regarding how different concrete UI technologies are being applied in the projects, solving the common requirements in the same way.

Architecture Views

As for the server we distinguish between the business and the technical architecture. Where the business architecture is different from project to project and relates to the concrete design of dialog components given concrete requirements, the technical architecture can be applied to multiple projects.

The focus of this document is to provide a technical reference architecture on the client on a very abstract level defining required layers and components. How the architecture is implemented has to be defined for each UI technology.

The technical infrastructure architecture is out of scope for this document and although it needs to be considered, the concepts of the reference architecture should work across multiple TI architecture, i.e. native or web clients.

2.1.2 OASP Reference Client Architecture

The following gives a complete overview of the proposed reference architecture. It will be built up incrementally in the following sections.

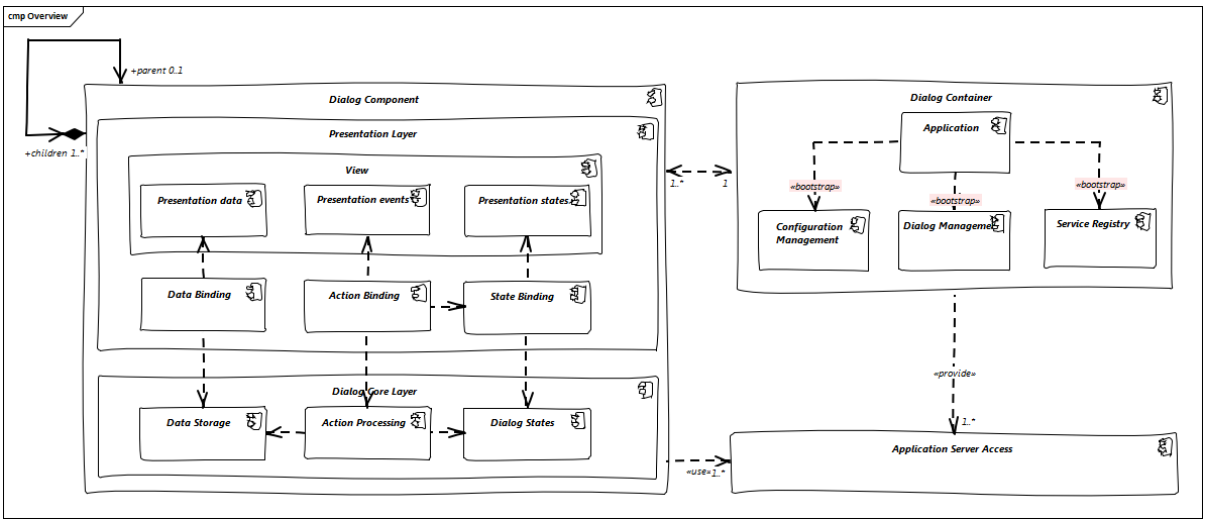


Figure 1 Overview

Client Architecture

On the highest level of abstraction we see the need to differentiate between dialog components and their container they are managed in, as well as the access to the application server being the backend for the client (e.g. an OASP4J instance). This section gives a summary of these components and how they relate to each other. Detailed architectures for each component will be supplied in subsequent sections

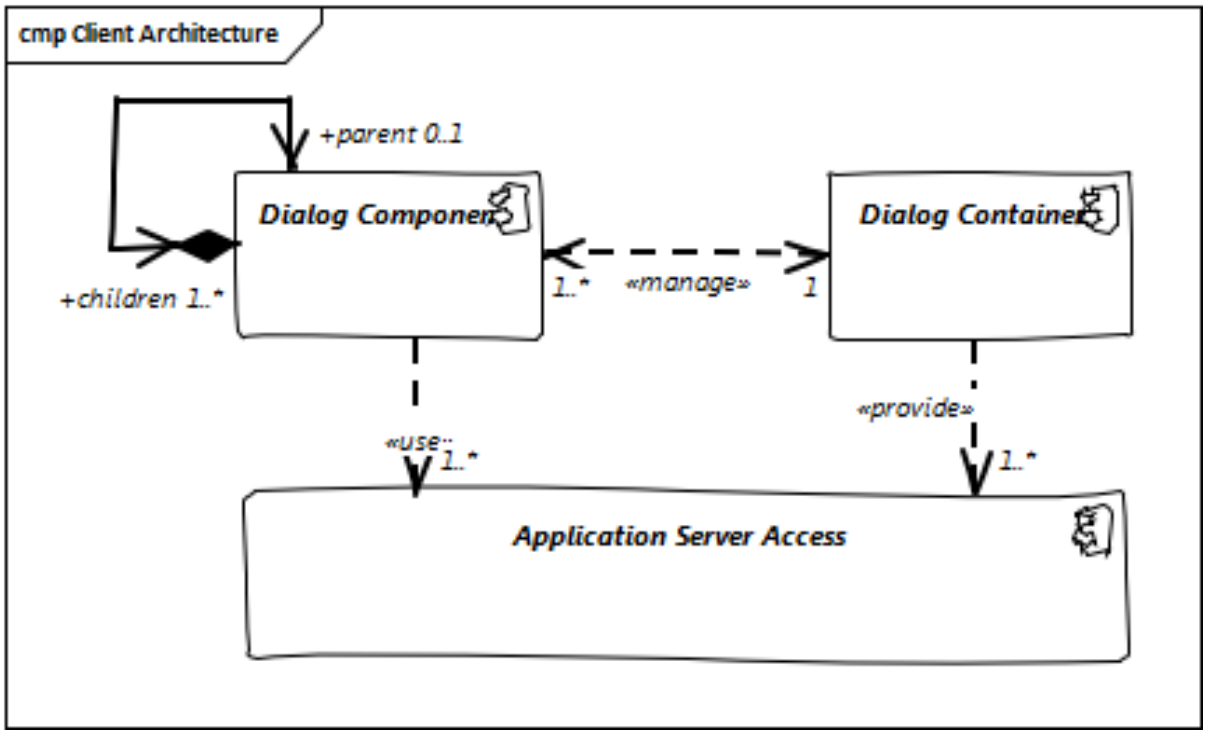


Figure 2 Overview of Client Architecture

Dialog Component

A dialog component is a logical, self-contained part of the user interface. It accepts user input and actions and controls communication with the user. Dialog components use the services provided by the

dialog container in order to execute the business logic. They are self-contained, i.e. they possess their own user interface together with the associated logic, data and states.

- Dialog components can be composed of other dialog components forming a hierarchy
- Dialog components can interact with each other. This includes communication of a parent to its children, but also between components independent of each other regarding the hierarchy.

Dialog Container

Dialog components need to be managed in their lifecycle and how they can be coupled to each other. The dialog container is responsible for this along with the following:

- Bootstrapping the client application and environment
  - Configuration of the client
  - Initialization of the application server access component
- Dialog Component Management
  - Controlling the lifecycle
  - Controlling the dialog flow
  - Providing means of interaction between the dialogs
  - Providing application server access
  - Providing services to the dialog components (e.g. printing, caching, data storage)
- Shutdown of the application

Application Server Access

Dialogs will require a backend application server in order to execute their business logic. Typically in an OASP application the service layer will provide interfaces for the functionality exposed to the client. These business oriented interfaces should also be present on the client backed by a proxy handling the concrete call of the server over the network. This component provides the set of interfaces as well as the proxy.

Dialog Container Architecture

The dialog container can be further structured into the following components with their respective tasks described in own sections:

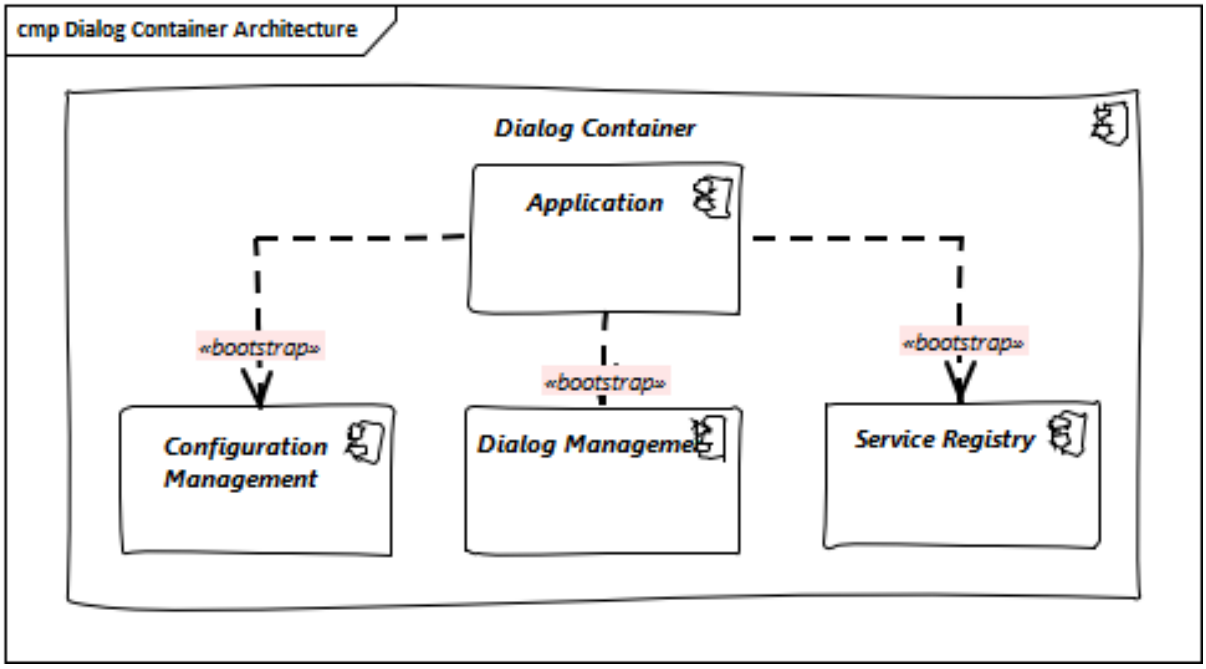


Figure 3 Dialog Container Architecture

Application

The application component represents the overall client in our architecture. It is responsible for bootstrapping all other components and connecting them with each other. As such it initializes the components below and provides an environment for them to work in.

Configuration Management

The configuration management manages the configuration of the client, so the client can be deployed in different environments. This includes configuration of the concrete application server to be called or any other environment-specific property.

Dialog Management

The Dialog Management component provides the means to define, create and destroy dialog components. It therefore offers basic lifecycle capabilities for a component. In addition it also allows composition of dialog components in a hierarchy. The lifecycle is then managed along the hierarchy, meaning when creating/destroying a parent dialog, this affects all child components, which are created/ destroyed as well.

Service Registry

Apart from dialog components, a client application also consists of services offered to these. A service can thereby encompass among others:

- Access to the application server
- Access to the dialog container functions for managing dialogs or accessing the configuration
- Dialog independent client functionality such as Printing, Caching, Logging, Encapsulated business logic such as tax calculation

- Dialog component interaction

The service registry offers the possibility to define, register and lookup these services. Note that these services could be dependent on the dialog hierarchy, meaning different child instances could obtain different instances / implementations of a service via the service registry, depending on which service implementations are registered by the parents.

Services should be defined as interfaces allowing for different implementations and thus loose coupling.

Dialog Component Architecture

A dialog component has to support all or a subset of the following tasks:
(T1) Displaying the user interface incl. internationalization
(T2) Displaying business data incl. changes made to the data due to user interactions and localization of the data
(T3) Accepting user input including possible conversion from e.g. entered Text to an Integer
(T4) Displaying the dialog state
(T5) Validation of user input
(T6) Managing the business data incl. business logic altering it due to user interactions
(T7) Execution of user interactions
(T8) Managing the state of the dialog (e.g. Edit vs. View)
(T9) Calling the application server in the course of user interactions

Following the principle of separation of concerns, we further structure a dialog component in an own architecture allowing us the distribute responsibility for these tasks along the defined components:

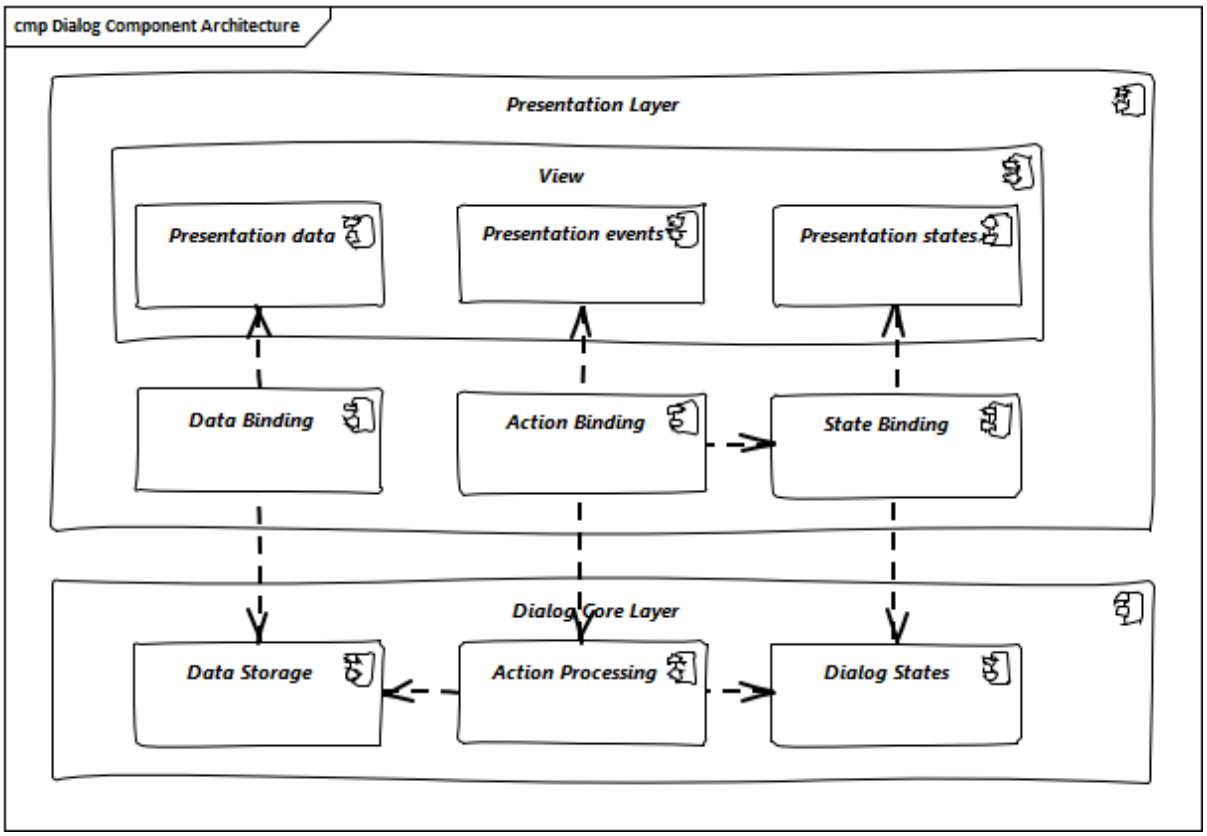


Figure 4 Overview of dialog component architecture

Presentation Layer

The presentation layer generates and displays the user interface, accepts user input and user actions and binds these to the dialog core layer (T1-5). The tasks of the presentation layer fall into two categories:

- **Provision of the visual representation (View component)**
  The presentation layer generates and displays the user interface and accepts user input and user actions. The logical processing of the data, actions and states is performed in the dialog core layer. The data and user interface are displayed in localized and internationalized form.

- **Binding of the visual representation to the dialog core layer**
  The presentation layer itself does not contain any dialog logic. The data or actions entered by the user are then processed in the dialog core layer. There are three aspects to the binding to the dialog core layer. We refer to ???data binding???, ???state binding??? and ???action binding???. Syntactical and (to a certain extent) semantic validations are performed during data binding (e.g. cross-field plausibility checks). Furthermore, the formatted, localized data in the presentation layer is converted into the presentation-independent, neutral data in the dialog core layer (parsing) and vice versa (formatting).

Dialog Core Layer

The dialog core layer contains the business logic, the control logic, and the logical state of the dialog. It therefore covers tasks T5-9:

- **Maintenance of the logical dialog state and the logical data**
  The dialog core layer maintains the logical dialog state and the logical data in a form which is independent of the presentation. The states of the presentation (e.g. individual widgets) must not be maintained in the dialog core layer, e.g. the view state could lead to multiple presentation states disabling all editable widgets on the view.

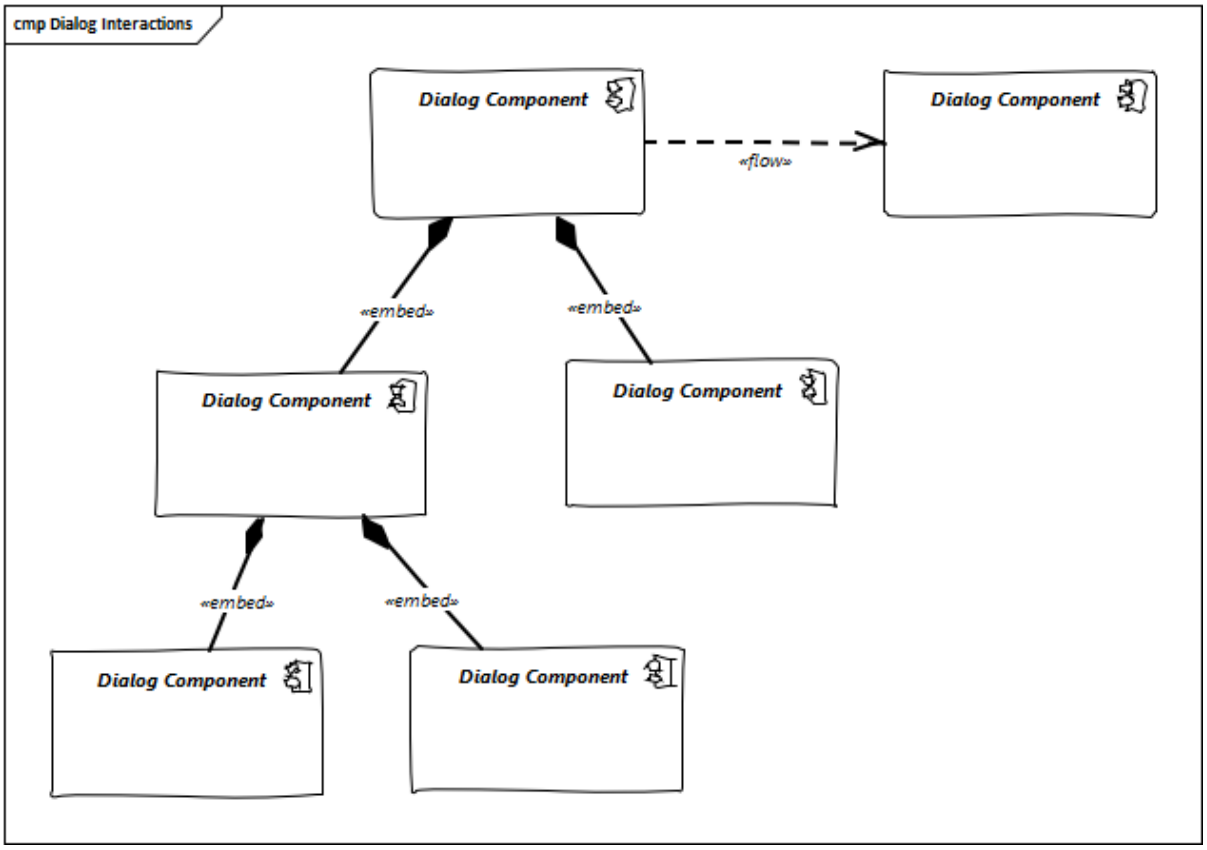
- **Implementation of the dialog and dialog control logic**
  The component parts in the dialog core layer implement the client specific business logic and the dialog control logic. This includes, for example, the manipulation of dialog data and dialog states as well as the opening and closing of dialogs.

- **Communication with the application server**
  The dialog core layer calls the interfaces of the application server via the application server access component services.

The dialog core layer should not depend on the presentation layer enforcing a strict layering and thus minimizing dependencies.

Interactions between dialog components

Dialog components can interact in the following ways:



- **Embedding of dialog components**
  As already said dialog components can be hierarchically composed. This composition works by embedding on dialog component within the other. Apart from the lifecycle managed by the dialog container, the embedding needs to cope for the visual embedding of the presentation and core layer.

  - **Embedding dialog presentation**
    The parent dialog needs to either integrate the embedded dialog in its layout or open it in an own model window.

  - **Embedding dialog core**
    The parent dialog needs to be able to access the embedded instance of its children. This allows initializing and changing their data and states. On the other hand the children might require context information offered by the parent dialog by registering services in the hierarchical service registry.

- **Dialog flow**
  Apart from the embedding of dialog components representing a tight coupling, dialogs can interact with each other by passing the control of the UI, i.e. switching from one dialog to another.

When interacting, dialog components should interact only between the same or lower layers, i.e. the dialog core should not access the presentation layer of another dialog component.

2.1.3 Appendix

Notes about Quasar Client

The Quasar client architecture as the consolidated knowledge of our CSD projects is the major source for the above drafted architecture. However, the above is a much simplified and more agile version thereof:

- Quasar Client tried to abstract from the concrete UI library being used, so it could decouple the business from the technical logic of a dialog. The presentation layer should be the only one knowing the concrete UI framework used. This level of abstraction was dropped in this reference architecture, although it might of course still make sense in some projects. For fast-moving agile projects in the web however introducing such a level of abstraction takes effort with little gained benefits. With frameworks like Angular 2 we would even introduce one additional seemingly artificial and redundant layer, since it already separates the dialog core from its presentation.

- In the past and in the days of Struts, JSF, etc. the concept of session handling was important for the client since part of the client was sitting on a server with a session relating it to its remote counterpart on the users PC. Quasar Client catered for this need, by very prominently differentiating between session and application in the root of the dialog component hierarchy. However, in the current days of SPA applications and the lowered importance of servers-side web clients, this prominent differentiation was dropped. When still needed the referenced documents will provide in more detail how to tailor the respective architecture to this end.

2.1.4 References

- Architecture Guidelines for Application Design: https://troom.capgemini.com/sites/vcc/engineering/Cross%20Cutting/ArchitectureGuide/Architecture_Guidelines_for_Application_Design_v2.0.docx

- Quasar Client Architekturen: https://troom.capgemini.com/sites/vcc/Shared%20Documents/CrossCuttingContent/TopicOrientedCCC/QuasarOverview/NCE%20Quasar%20Review%20Workshop%202009-11-17/Quasar%20Development/Quasar-Client-Architectures.doc

2.2 OASP Angular Architecture

This document defines how the Angular (v2) framework implements the OASP Client Reference Architecture mapping its concepts to those of OASP. Since Angular 2 provides often multiple ways of implementing them, this document also defines the preferred and consistent way for realization of the required components..

We assume Angular CLI and webpack to be used when developing such applications.

2.2.1 Mapping to OASP Client Architecture

Angular Architecture Summary

The architecture of Angular is summarized very thoroughly here:

https://angular.io/docs/ts/latest/guide/architecture.html

As an orientation the following figure of that page is repeated here:

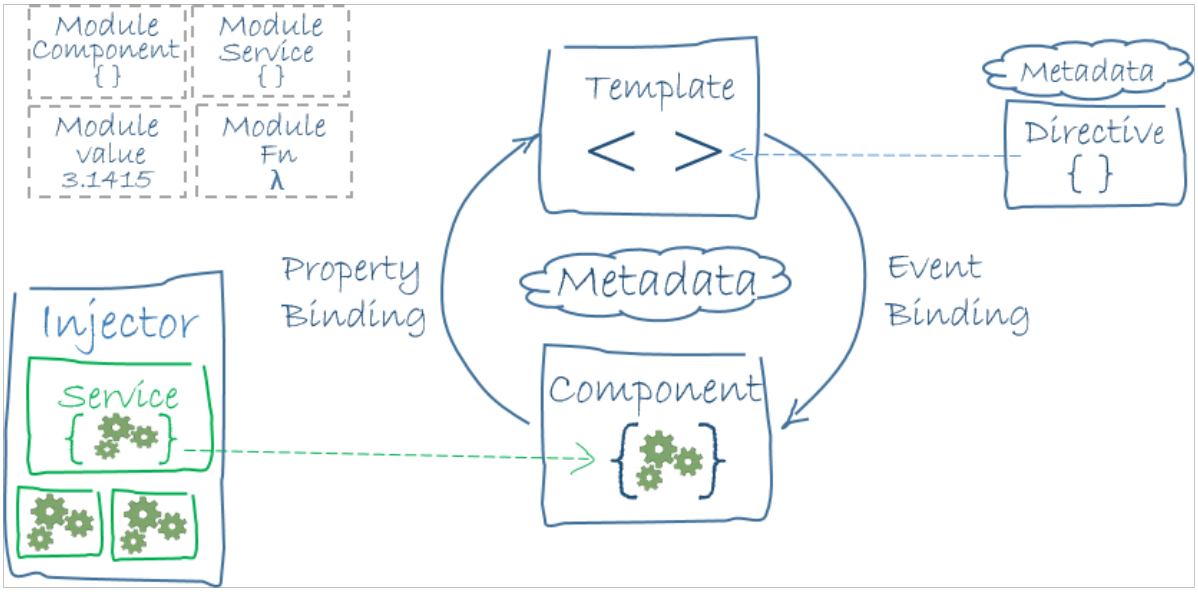


Figure 1 Angular Architcture (Source: s. link above)

High Level Architecture

Component	Implementation in Angular
Dialog Component	Each dialog is developed as an Angular component already supporting the idea of presentation and core layer by the use of templates and component class (s. below).
Dialog Container	The dialog container is provided by the Angular framework itself represented by the root module. It already covers most of the concepts of the reference architecture natively.

Component	Implementation in Angular
Application Server Proxy	Each REST service provided by an OASP4J server should be represented by a concrete Typescript interface marked as an injectable Angular service. CobiGen is to be used for generation of these interfaces. The complete application server access component should be bundled in one NgModule providing access to the complete service layer of the application server.

Dialog Container

Component	Implementation in Angular
Application	The Angular root component represents the application component covering the bootstrapping and the Angular framework.
Configuration Management	An Environment Constant class will be provided for each environment. This is part of webpack and allows us to define environment-specific variables, which are bundled during the build for the concrete environment.
Service Registry	The Angular Injector natively implements this component. It allows to define services which are wired to the components during initialization.
Dialog Management	Again covered by the Angular root module defining all dialog components in the declaration array also allowing nesting these components within each other.

Dialog Components

Each dialog component corresponds to one Angular component (s. above). The presentation layer is implemented by the component???s template and the dialog core by the component class.

Presentation Layer

Component	Implementation in Angular
View	HTML template of the angular component
Presentation data	Data nodes of the DOM tree.
Presentation state	Attribute nodes of the DOM tree steering the visualization
Presentation events	DOM events
Data Binding	Native angular support by using:

Component	Implementation in Angular
	- Two-way data binding for all user input - Interpolation or property binding for read-only values (e.g. dynamic titles)
Action Binding	Native angular support by using event binding.
State Binding	Native angular support by using two-way data binding

Dialog Core Layer

Component	Implementation in Angular
Data Storage	Own typescript class generated from transfer objects via CobiGen.
Action Processing	Dialog core represented by the class of the Angular component.
Dialog state	Own typescript class generated from transfer objects via CobiGen.
Interaction of dialog components	Angular already natively supports all required interactions as described in more detail here: https://angular.io/docs/ts/latest/cookbook/component-communication.html
Embedding	Dialog components can be nested by simply specifying their selector in the parent view. Data can be passed and returned by Input and Output parameters. In addition each dialog core can access its children via the ViewChild decorator.
Dialog Flow	The angular router implements the flow between dialog components.

Other concepts

Angular 2 provides many more concepts, not directly required by our client architecture:

- Directives
- Modules
- ????

TODO: define in more detail how they relate to our architecture and how they should be used.

2.2.2 Angular Usage Conventions and Best Practices

Dialog Component

For very simple dialog inlining the template dialog is permitted. For medium to complex dialogs, an explicit template HTML file is required.

Service Registry

Angular allows an hierarchical dependency injection out of the box even with the possibility to abstract from the concrete implementation being wired:

https://angular.io/docs/ts/latest/guide/dependency-injection.html

Note however, that Typescript interfaces cannot be used to define the service abstraction, since they are not present during runtime.

TODO: define concrete usage convention

Data Binding

TODO: when to use interpolation and when property binding?

State Binding

TODO

Action Binding

TODO

2.2.3 Packaging Conventions

TODO:

- One big project for client vs. one project per dialog component
- How do NgModules relate to dialog components? 1:1 or 1:n relation?
- One NgModule per component on the server side?
- One NgModule per project or multiple?

2.2.4 Directory Conventions

- We follow the webpack conventions for an angular project.
- Introduce directories for layers or dialogs?